Limbaje pentru programarea concurentă

- Includ *facilități speciale* în vederea descrierii unor procese paralele şi concurente, precum şi a sincronizării şi comunicării între ele.

- Unele dintre ele se adresează cu precădere unui anumit domeniu cum ar fi **programarea în timp real, programarea sistemelor distribuite** sau, mai general, *programarea sistemelor încorporate* (*embedded systems*).

- Aplicațiile în timp real cer ca sistemul să răspundă la un eveniment exterior, într-un interval de timp finit. Este necesar deci ca programul să aibă acces la un ceas de timp real, fie pentru a asigura întârzierea unui proces pe o anumită perioadă de timp fie pentru a declanşa un eveniment (proces) la o anumită oră.

Problema filozofilor la masă

Cinci filozofi îşi petrec viața mâncând şi gândind. Când unui filozof îi este foame, intră în sufragerie, se aşează la masă şi mănâncă. Pentru a se servi din vasul cu spaghetti, plasat în mijlocul mesei, şi a mânca, el are nevoie de două furculițe. Pe masă se află doar cinci furculițe, câte una între fiecare pereche de locuri. Fiecare filozof are acces doar la furculița din stânga şi din dreapta lui. După ce a reuşit să mănânce (activitate de durată finită), filozoful pune furculițele la locul lor, apoi se scoală şi părăseşte sufrageria.

Să descriem *comportamentul filozofilor (procesele*) între care sunt *partajate furculițele (resursele*), utilizând limbajul Edison.

Filozofii aşteaptă până în momentul în care sunt libere ambele furculițe, intrând în posesia lor printr-o singură operație. Tabloul *furculite* indică în fiecare moment câte furculițe stau la dispoziția fiecărui filozof. Există situația nefavorabilă ca un anumit filozof să „moară de foame”, în cazul în care vecinii din stânga şi respectiv din dreapta mănâncă în mod alternativ.

Cei 5 filozofi reprezintă 5 activări ale procedurii *viatafilozof* fiecare în câte o ramură a instrucțiunii **cobegin**, care lansează în lucru procesele paralele şi concurente.

Exemplu EDISON

```
proc filozofi
    module
        array tfurculite[0..4] (int)
        var furculite: tfurculite;  i:int;
        proc filodr(i: int): int
        begin val filodr:= (i+1) mod 5 end
        proc filost(i:int): int
        begin if i = 0 do val filost := 4
                  else true val filost := i-1
        end  "filost"
        *proc ridica(filo: int)
        begin when furculite[filo] = 2 do
                    furculite[filodr(filo)] := furculite[filodr(filo)]-1;
                    furculite[filost(filo)] := furculite[filost(filo)]-1;
              end   "when"
        end   "ridica"
        *proc depune(filo: int)
        begin when true do "when asigura accesul exclusiv la tabelul furculite"
                    furculite[filodr(filo)] := furculite[filodr(filo)]+1;
                    furculite[filost(filo)] := furculite[filost(filo)]+1;
              end   "when"
        end   "depune"
    begin  "initializarea variabilelor modulului"
        i := 0;
        while i<5 do
            furculite[i] := 2;   i := i+1
        end   "while"
    end  "module"
    proc viatafilozof(i: int)
    begin
        while true do
                - gandeste -
                ridica(i);
                - mananca -
                depune(i);
        end   "while"
end  "viatafilozof"
begin
    cobegin
        1 do viatafilozof(0) also 2 do viatafilozof(1)
        also 3 do viatafilozof(2) also 4 do viatafilozof(3)
        also 5 do viatafilozof(4)
```

```
        end  “cobegin”
end  “filozofi”
```

Programarea sistemelor distribuite

Un **sistem distribuit** este o colecție de calculatoare(rețea) capabile de a *schimba informații* între ele. Elementele colecției poartă numele de **noduri**.

Un astfel de sistem poate fi programat pentru a rezolva probleme de *concurență ş*i de *paralelism.*

Probleme algoritmice tipice care se pun pentru astfel de rețele:

- *Sincronizarea*: aşteptarea îndeplinirii unei condiții;
- *Difuzarea totală (broadcasting)*: transmiterea unui mesaj către toate nodurile;
- *Selectarea unui proces* pentru îndeplinirea anumitor acțiuni;
- *Detectarea terminării*: un nod ce întreprinde o acțiune, în care antrenează şi alte noduri, trebuie să fie capabil să detecteze momentul încheierii acțiunii;
- *Excluderea reciprocă*: utilizarea unor resurse sub excludere reciprocă (de exemplu modificarea unui fişier sau scrierea la imprimantă);
- *Detectarea şi prevenirea blocării totale (deadlock);*
- *Gestionarea distribuită a fişierelor.*

Limbajele specializate în programarea aplicațiilor pentru astfel de sisteme trebuie să includă facilități care să rezolve direct problemele de mai sus. Un astfel de limbaj este JAVA.

Exemplu. *Ne propunem să realizăm un chat (o conversație) între două calculatoare*

Aplicația utilizează **modelul Client/Server**, aplicat pe scară largă în sistemele distribuite şi care constă în:

- O mulțime de *procese Server* care gestionează resurse;
- O mulțime de *procese Client* care necesită acces la resursele furnizate de servere.

Pentru simplificare, considerăm că fiecare mesaj trimis de la un calculator la celălalt se reduce la o linie de text.

Serverul este primul care trebuie pornit. Apoi clientul este cel care începe discuția, transmițând un mesaj şi aşteptând să primească mesajul de răspuns, activități care se repetă până când clientul trasmite mesajul „STOP”. Serverul repetă şi el succesiunea de activități recepție+transmisie până când primeşte mesajul „STOP”.

- Clientul va folosi unitatea de compilare *Client.java*;
- Serverul va folosi unitatea de compilare *Server.java*.

➢ **Observație**. Metoda *readLine* a clasei *DataInputStream* este considerată „depreciată” (*deprecated*), deoarece în unele situații nu asigură o conversie corectă octet -> caracter; de aceea la compilare apare un mesaj de

avertizare corespunzător. Este indicat să apelăm la transmiterea datelor la nivel de caracter dar, pentru simplificarea expunerii, nu vom proceda în acest mod.

```
// Unitatea de compilare Client.java
import java.net.*; import java.io.*;
class Client {
  public static void main (String[] sir) throws IOException {
    Socket cs=null;
    DataInputStream is = null;
    DataOutputStream os = null;
    try {
      cs = new Socket("localhost",5678);
      is = new DataInputStream(cs.getInputStream());
      os = new DataOutputStream(cs.getOutputStream());
    }
    catch (UnknownHostException e) {
      IO.writeln("Host inexistent");
    }

    DataInputStream stdin = new DataInputStream(System.in);
    String linie;
    for(;;) {
      linie = stdin.readLine()+"\n";
      os.writeBytes(linie);
      IO.writeln("Transmisie :\t" + linie);
      if (linie.equals("STOP\n")) break;
      linie = is.readLine();
      IO.writeln("Receptie :\t" + linie);
    }

    IO.writeln("GATA");
```

```
    cs.close(); is.close(); os.close();
  }
}
```

```
// Unitatea de compilare Server.java
import java.net.*; import java.io.*;
class Server {
  public static void main(String[] sir) throws IOException {
    ServerSocket ss = null;
    Socket cs = null;
    DataInputStream is = null;
    DataOutputStream os = null;
    ss = ServerSocket(5678);
    IO.writeln("Serverul a pornit !");
    cs = ss.accept();
    is = new DataInputStream(cs.getInputStream());
    os = new DataOutputStream(cs.getOutputStream());

    DataInputStream stdin = new DataInputStream(System.in);
    String linie;
    for(;;) {
      linie = is.readLine();
      IO.writeln("Receptie :\t" + linie);
      if (linie.equals("STOP")) break;
      linie = stdin.readLine()+"\n"; os.writeBytes(linie);
    }
    cs.close(); is.close(); os.close();
  }
}
```

Socket-uri

O adresă IP identifică un calculator în Internet. Un număr de port identifică un program care rulează pe un calculator. O combinație dintre un număr de port şi o adresă IP determină un *socket* şi furnizează un punct final unei căi de comunicare.

Atunci când două aplicații trebuie să comunice, ele trebuie să se găsească reciproc în Internet. De obicei este responsabilitatea clientului să găsească serverul. Un client încearcă să se conecteze la un server prin inițializarea unei *conexiuni de socket*. Primul mesaj pe care o aplicație client îl transmite unui server conține socketul clientului. Serverul trasmite propria adresă de socket către client în primul său mesaj de răspuns.

Transmiterea datelor se realizează prin intermediul fluxurilor de intrare/ieşire asociate socketurilor. Fluxurile se obțin prin metodele getInputStream şi getOutputStream ale clasei Socket.